Algoritmi e Strutture Dati

Modelli di calcolo e analisi di algoritmi

Maria Rita Di Berardini, Emanuela Merelli[1]

[1]Dipartimento di Matematica e Informatica
Università di Camerino

Analisi di Algoritmi

Analizzare un algoritmo vuol dire determinare le risorse necessarie all'algoritmo in termini di

spazio di memoria
(quantità di memoria utilizzata durante l'esecuzione)
e
tempo computazionale
(tempo di esecuzione)

Analisi di Algoritmi

L'analisi della complessità di un algoritmo in termini di tempo di esecuzione consente di:

- stimare il tempo impiegato
- stimare il più grande input gestibile in termini ragionevoli
- confrontare l'efficienza di due algoritmi diversi
- ottimizzare le parti "critiche"

Definiamo una funzione T : **dimensione** $\longrightarrow$ **tempo** impiegato

Dimensione dell'input

Per molti problemi (ex. l'ordinamento) la misura più naturale è il numero di elementi (criteri di costo uniforme)

Per altri (ex. moltiplicazione di numeri interi) la misura migliore è il numero totale di bit necessari per la rappresentazione dell'input (criteri di costo logaritmico)

In realtà ciascun elemento è rappresentato da un numero costante di bit, quindi le due misure coincidono a meno di una costante moltiplicativa

In altri casi ancora, è più appropriato descrivere la dimensione con due numeri; ex: se l'input è una matrice bidimensionale la dimensione dell'input è $\#righe \times \#colonne$

Modello di Calcolo

Prima di analizzare un algoritmo abbiamo bisogno di stabilire quale sarà la tecnologia di riferimento utilizzata per eseguire gli algoritmi quando saranno realizzati come programmi

Assumiamo di utilizzare

Mono-Processore + RAM (Random Access Memory)

assenza di concorrenza e parallelismo

Definizione di tempo

Tempo = "wall-clock" time ossia il tempo tempo effettivamente impiegato per eseguire un algoritmo

Dipende da troppi fattori (non sempre prevedibili)

1. bravura del programmatore
2. linguaggio di programmazione utilizzato
3. processore, memoria (cache, primaria, secondaria)
4. sistema operativo, processi attualmente in esecuzione

Dobbiamo usare un modello astratto: introduciamo un concetto di tempo legato al "# di operazioni elementari" o di passi eseguiti per il calcolo dell'output corrispondente

Tempo di esecuzione

Numero di operazioni elementari o "passi" eseguiti per il calcolo dell'output

Numero di operazioni elementari o "passi" eseguiti per il calcolo dell'output

passo $\cong$ una linea di pseudocodice

Hp: ogni passo riferito ad una linea i, è eseguito in un tempo costante c_i

Parte I

Il problema dell'ordinamento

Il problema dell'ordinamento

Definition

Dato un insieme di n numeri $\langle a_1, a_2, \ldots, a_n \rangle$, trovare un'opportuna permutazione $\langle a'_1, a'_2, \ldots, a'_n \rangle$ tale che $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Input: $\langle a_1, a_2, \ldots, a_n \rangle$

Output: $\langle a'_1, a'_2, \ldots, a'_n \rangle$ oppure

$\langle a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)} \rangle$

dove π è un'opportuna permutazione degli indici $1, \ldots, n$

Istanza del problema

Input: (31, 41, 59, 26, 41, 58)

Output: (26, 31, 41, 41, 58, 59)

La scelta del migliore algoritmo dipende:

- dal numero di elementi da ordinare
- da quanto gli elementi siano già ordinati
- dispositivo di memoria (metodo d'accesso)

Idea per ordinare

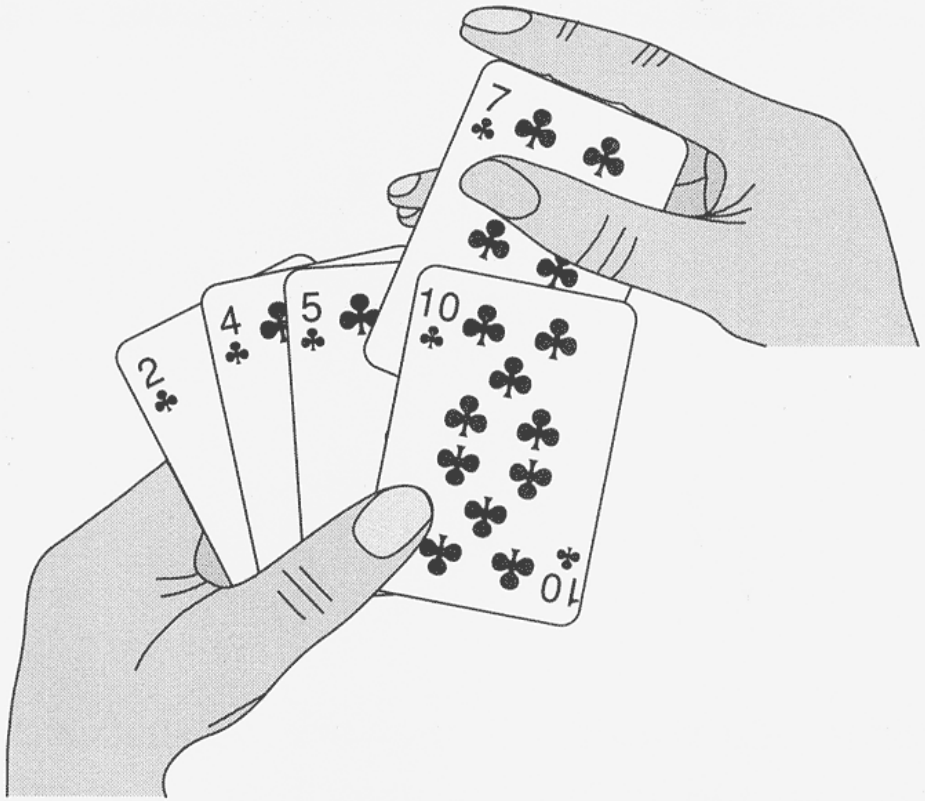
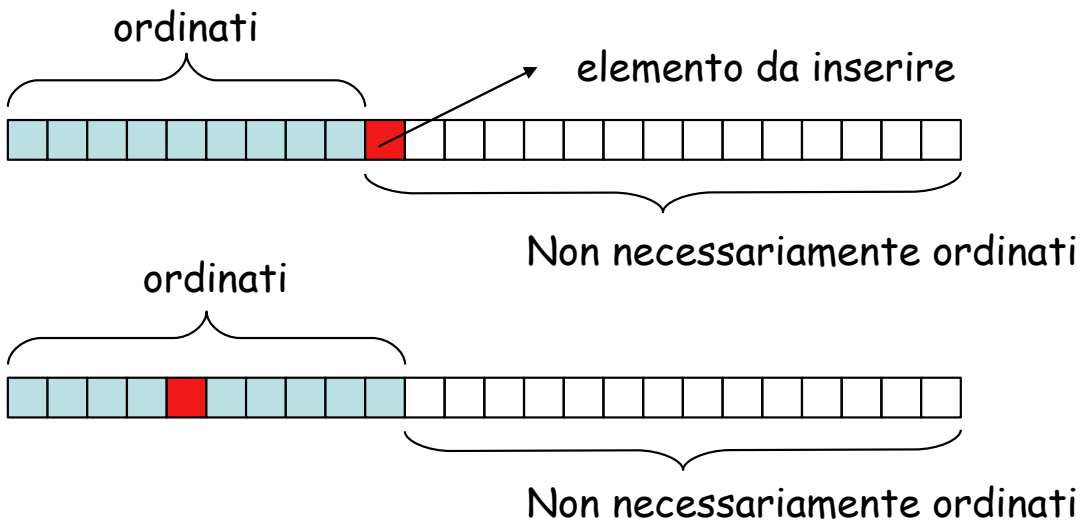


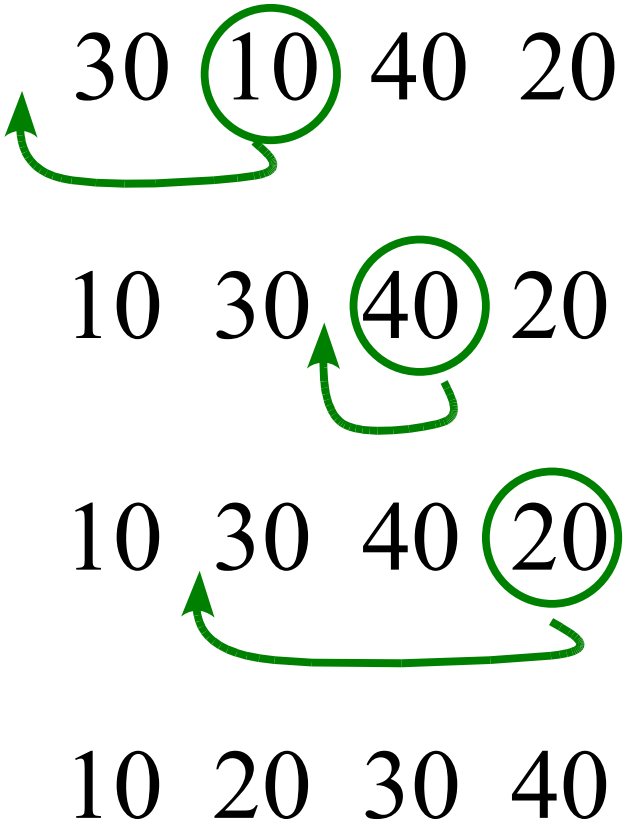
Figure 2.1 Sorting a hand of cards using insertion sort.

Idea per ordinare

Ad ogni passo si ha una sottosequenza ordinata in cui inserisco un nuovo elemento dell'input:



Idea per ordinare



Insertion Sort

L'algoritmo di ordinamento Insertion Sort risulta efficiente nel caso in cui il numero di elementi (n) da ordinare è piccolo

```
for j ← 2 to lenght[A]
    do key ← A[j]
        ▷ Si inserisce A[j] nella sequenza ordinata A[1..j-1]
        i ← j-1
        while i > 0 e A[i] > key
            do   A[i+1] ← A[i]
                 i ← i-1
        A[i+1] ← key
```

Istanza $A = \{5, 2, 4, 6, 1, 3\}$

$$
\begin{array}{lcccccc}
j = 2 & 5 & 2 & 4 & 6 & 1 & 3 \\
j = 3 & 2 & 5 & 4 & 6 & 1 & 3 \\
j = 4 & 2 & 4 & 5 & 6 & 1 & 3 \\
j = 5 & 2 & 4 & 5 & 6 & 1 & 3 \\
j = 6 & 1 & 2 & 4 & 5 & 6 & 3 \\
j = 7 & 1 & 2 & 3 & 4 & 5 & 6
\end{array}
$$

Insertion Sort

Il tempo computazionale impiegato dalla procedura Insertion Sort dipende

- dalla dimensione input, il numero degli elementi
- dallo ordinamento implicito nella sequenza

Il tempo di esecuzione è espresso in funzione della dimensione del problema, cioè dell'input

Insertion Sort

Insertion-sort(A,n)

statement	**costo**
1. for j = 2 to n {	c_1
2. key = A[j]	c_2
3. i = j-1	c_3
4. while $(i > 0)$ and $(A[i] > key)$ {	c_4
5. $A[i+1] = A[i]$	c_5
6. $i = i - 1$	c_6
7. }	
8. $A[i+1] = key$	c_8
9. }	

Insertion Sort

Indichiamo con n_i il numero di volte che un passo i viene eseguito:

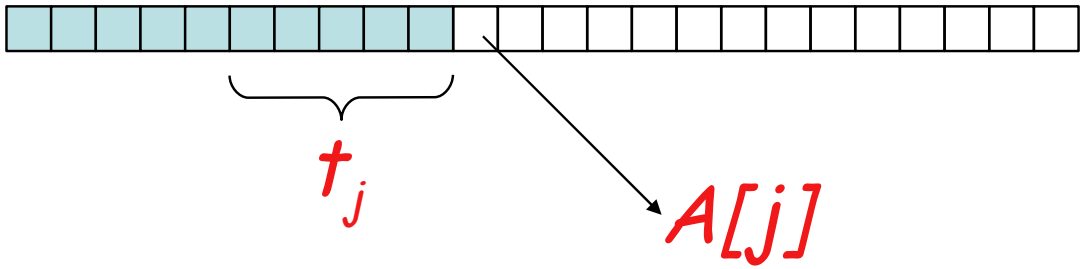
$$T(n) = n_1 c_1 + n_2 c_2 + \ldots c_k n_k = \sum_{i=1}^{k} c_i n_i$$

Insertion-sort

Insetion-sort(A,n)

statement	**costo**	**volte**
1. for j = 2 to n {	c_1	n
2. key = A[j]	c_2	$n-1$
3. i = j-1	c_3	$n-1$
4. while $(i > 0)$ and $(A[i] > key)$ {	c_4	??
5. $A[i+1] = A[i]$	c_5	??
6. $i = i-1$	c_6	??
7. }		
8. $A[i+1] = key$	c_8	$n-1$
9. }		

Insertion Sort: analisi del costo computazionale



Fissato $j = 2, \ldots n$, denotiamo con t_j numero di elementi maggiori di $key = A[j]$ (t_j il numero di volte che il test del ciclo while viene eseguito, $t_j = t_j + 1$). Il valore di t_j dipende dal vettore in input:

- Caso ottimo (vettore già ordinato) $t_j = 0$ e $t_j = 1$
- Caso peggiore (ordinato in maniera decrescente) $t_j = j - 1$ e $t_j = j$
- Caso medio $t_j = j/2$

Insertion Sort: analisi del costo computazionale

Insetion-sort(A,n)

statement	**costo**	**volte**
1. for j = 2 to n {	c_1	n
2. key = A[j]	c_2	$n-1$
3. i = j-1	c_3	$n-1$
4. while $(i > 0)$ and $(A[i] > key)$ {	c_4	n_4
5. $A[i+1] = A[i]$	c_5	n_5
6. $i = i-1$	c_6	n_6
7. }		
8. $A[i+1] = key$	c_8	$n-1$
9. }		

$$n_4 = \sum_{j=2}^{n} t_j, \text{ e } n_5 = n_6 = \sum_{j=2}^{n}(t_j - 1)$$

Insertion Sort: analisi del costo computazionale

Insetion-sort(A,n)

statement	**costo**	**volte**
1. for j = 2 to n {	c_1	n
2. key = A[j]	c_2	$n-1$
3. i = j-1	c_3	$n-1$
4. while $(i > 0)$ and $(A[i] > key)$ {	c_4	$\sum_{j=2}^{n} t_j$
5. $A[i+1] = A[i]$	c_5	$\sum_{j=2}^{n}(t_j - 1)$
6. $i = i - 1$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7. }		
8. $A[i+1] = key$	c_8	$n-1$
9. }		

$$T(n) = c_1 n + (c_2 + c_3 + c_8)(n-1) + c_4 \sum_{j=2}^{n} t_j + (c_5 + c_6) \sum_{j=2}^{n} (t_j - 1)$$

Insertion Sort: analisi del costo computazionale nel caso ottimo

Nel **caso ottimo** $t_j = 1$, quindi

$$\begin{aligned} T(n) = & \quad c_1 n + (c_2 + c_3 + c_8)(n-1) + c_4 \sum_{j=2}^{n} t_j + (c_5 + c_6) \sum_{j=2}^{n} (t_j - 1) \\ = & \quad c_1 n + (c_2 + c_3 + c_8)(n-1) + c_4 \sum_{j=2}^{n} 1 \\ = & \quad c_1 n + (c_2 + c_3 + c_8)(n-1) + c_4 (n-1) \\ = & \quad (c_1 + c_2 + c_3 + c_4 + c_8) n - (c_2 + c_3 + c_4 + c_8) \end{aligned}$$

$$T(n) = an + b$$

dove a e b sono delle costanti

$T(n)$ è una funzione lineare in n

Insertion Sort: analisi del costo computazionale nel caso peggiore

Nel **caso peggiore** $t_j = j$, quindi

$$\begin{aligned} T(n) = & \quad c_1 n + (c_2 + c_3 + c_8)(n-1) + c_4 \sum_{j=2}^{n} t_j + (c_5 + c_6) \sum_{j=2}^{n} (t_j - 1) \\ = & \quad c_1 n + (c_2 + c_3 + c_8)(n-1) + c_4 \sum_{j=2}^{n} j + (c_5 + c_6) \sum_{j=2}^{n} (j - 1) \end{aligned}$$

Fact

$$\sum_{j=2}^{n} j = (\sum_{j=1}^{n} j) - 1 = \frac{n(n+1)}{2} - 1 = \frac{n^2}{2} + \frac{n}{2} - 1$$

$$\sum_{j=2}^{n} j = \sum_{j=1}^{n-1} j = \frac{(n-1)n}{2} = \frac{n^2}{2} - \frac{n}{2}$$

Insertion Sort: analisi del costo computazionale nel caso peggiore

$$\begin{aligned} T(n) = & \quad c_1 n + (c_2 + c_3 + c_8)(n-1) + c_4 \sum_{j=2}^{n} j + (c_5 + c_6) \sum_{j=2}^{n} (j-1) \\ = & \quad c_1 n + (c_2 + c_3 + c_8)(n-1) + c_4 (\tfrac{n^2}{2} + \tfrac{n}{2} - 1) + (c_5 + c_6)(\tfrac{n^2}{2} - \tfrac{n}{2}) \\ = & \quad \tfrac{c_4 + c_5 + c_6}{2} n^2 + (c_1 + c_2 + c_3 + \tfrac{c_4 - c_5 - c_6}{2} + c_8) n - (c_2 + c_3 + c_4 + c_8) \end{aligned}$$

$$T(n) = an^2 + bn + c$$

dove a, b e c sono delle costanti

$T(n)$ è una funzione quadratica in n

Analisi nel caso medio

L'analisi del tempo di esecuzione nel **caso medio** analizza come si comporta l'algoritmo in "media"

È legato al fatto che in alcune circostanze non è necessario che un algoritmo operi con una quantità limitata di risorsa per ogni possibile input, ma è sufficiente che sia "generalmente efficiente"

Per ogni n, il costo dell'algoritmo viene calcolato come la somma dei costi per ogni possibile input di dimensione n tenendo conto della probabilità che ciascun singolo caso si verifichi

$$T_{medio}(n) = \sum_{\{I:|I|=n\}} T(I)Pr(I)$$

Analisi nel caso medio

Bisogna determinare la distribuzione di probabilità di ciascun input

Problema: non sempre si dispone delle informazioni necessarie per determinare la distribuzione Pr

Una possibile soluzione: fare delle assunzioni plausibili sulle caratteristiche del problema

Ad esempio, nel caso del Insersion sort è plausibile assumere che, mediamente, solo la metà degli elementi della porzione di vettore $A[1...j-1]$ sia maggiore di *key*, e quindi che $t_j = j/2$

Insertion Sort: analisi del costo computazionale nel caso medio

Nel **caso medio** $t_j = j/2$, quindi

$$\begin{aligned} T(n) = & \quad c_1 n + (c_2 + c_3 + c_8)(n-1) + c_4 \sum_{j=2}^{n} t_j + (c_5 + c_6) \sum_{j=2}^{n} (t_j - 1) \\ = & \quad c_1 n + (c_2 + c_3 + c_8)(n-1) + c_4 \sum_{j=2}^{n} j/2 \\ & + (c_5 + c_6) \sum_{j=2}^{n} (j/2 - 1) \end{aligned}$$

$$\sum_{j=2}^{n} j/2 = \frac{1}{2} \sum_{j=2}^{n} j = \frac{1}{2} (\sum_{j=1}^{n} j - 1) = \frac{1}{2} (\frac{n^2}{2} + \frac{n}{2} - 1) = \frac{n^2}{4} + \frac{n}{4} - \frac{1}{2}$$

$$\sum_{j=2}^{n} (j/2 - 1) = \sum_{j=2}^{n} \frac{j-2}{2} = \frac{1}{2} \sum_{j=2}^{n} j - 2 = \frac{1}{2} \sum_{j=0}^{n-2} j = \frac{1}{2} \sum_{j=1}^{n-2} j =$$

$$\frac{1}{2} \frac{(n-2)(n-1)}{2} = \frac{1}{2} \frac{n^2 - 3n + 2}{2} = \frac{n^2}{4} - \frac{3n}{4} + \frac{1}{2}$$

Insertion Sort: analisi del costo computazionale nel caso medio

$$
\begin{aligned}
T(n) = \quad & c_1 n + (c_2 + c_3 + c_8)(n-1) + c_4 \sum_{j=2}^{n} j/2 \\
& + (c_5 + c_6) \sum_{j=2}^{n} (j/2 - 1) \\
= \quad & c_1 n + (c_2 + c_3 + c_8)(n-1) + c_4 (\tfrac{n^2}{4} + \tfrac{n}{4} - \tfrac{1}{2}) + \\
& (c_5 + c_6)(\tfrac{n^2}{4} - \tfrac{3n}{4} + \tfrac{1}{2}) \\
= \quad & \tfrac{c_4 + c_5 + c_6}{4} n^2 + (c_1 + c_2 + c_3 + \tfrac{c_4 - 3c_5 - 3c_6}{2} + c_8) n + \\
& (\tfrac{c_5 + c_6}{2} - (c_2 + c_3 + \tfrac{c_4}{2} + c_8))
\end{aligned}
$$

$$T(n) = an^2 + bn + c$$

dove a, b e c sono delle costanti

$T(n)$ è una funzione quadratica in n

Analisi del caso peggiore vs. analisi del caso medio

Il tempo di esecuzione nel caso ottimo non è molto significativo

Di solito prenderemo in considerazione il tempo di esecuzione nel caso peggiore. Perchè??

- il tempo di esecuzione nel caso peggiore rappresenta una limitazione superiore sui tempi di esecuzione per qualsiasi input, non possiamo fare di peggio
- per alcuni algoritmi il caso peggiore si verifica abbastanza spesso (ex. ricerca di un elemento in una base di dati)
- il caso medio è in generale altrettanto cattivo quanto il caso peggiore (vedi Insertion Sort)

Parte II

Analisi asintotica

Un graduale processo di astrazione

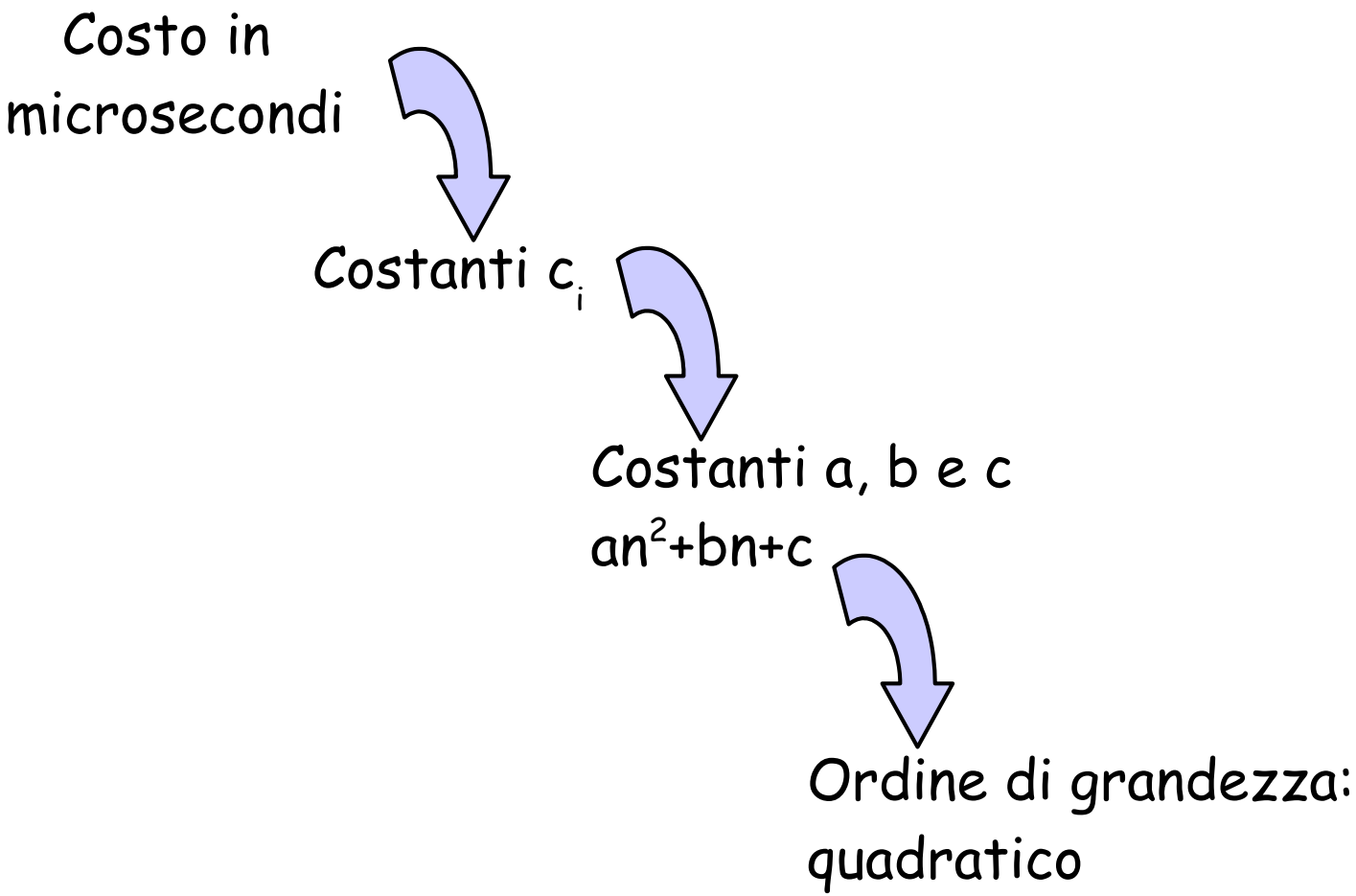
Passo 1: abbiamo ignorato il costo effettivo delle singole istruzioni introducendo delle costanti $(c_1, ..., c_8)$

Passo 2: successivamente ci siamo resi conto che anche queste costanti forniscono dettagli non necessari

- Insertion Sort: il tempo di esecuzione nel caso peggiore è $an^2 + bn + c$, dove a, b e c sono altre costanti che dipendono dai costi c_i

Passo 3: consideriamo solo l'ordine di grandezza - tasso di crescita - del tempo di esecuzione

Un graduale processo di astrazione



Analisi Asintotica

- Supponiamo di aver ricavato che il tempo (o il numero di passi) che sono necessari a completare un algoritmo sia $T(n) = 4n^2 - 2n + 2$
- Per grandi valori di n, il termine $4n^2$ diventerà preponderante rispetto agli altri, che potranno non essere considerati
  - per esempio, per n pari a 500, il termine $4n^2$ sarà pari a 1000 volte il termine $2n$
- Anche i coefficienti diventano irrilevanti se compariamo $T(n)$ con una di ordine superiore, come n^3 oppure 2^n
  - Anche se $T(n) = 1.000.000n^2$, $U(n) = n^3$ sarà maggiore di $T(n)$ per ogni n maggiore di 1.000.000 ($T(1.000.000) = 1.000.000^3 = U(1.000.000)$)

Analisi Asintotica

- Astraiamo: da alcuni dettagli irrilevanti (ex: termini di ordine diverso, costanti moltiplicative) e ci interessiamo solo al tasso di crescita - analizziamo come il tempo di esecuzione cresce asintoticamente in funzione della dimensione dell'input

- Obiettivo
  - semplificare l'analisi del tempo di esecuzione di un algoritmo prescindendo dai dettagli implementativi o di altro genere
  - classificare le funzioni in base al loro comportamento asintotico

- Asintoticamente non significa per tutti gli input. Esempio: input di piccole dimensioni

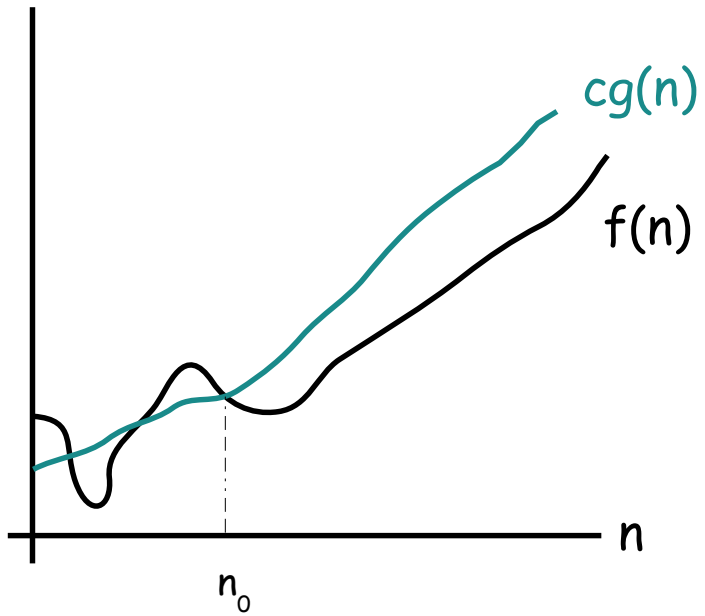
Analisi Asintotica

Importiamo alcune notazioni dalla matematica:

- O (o grande) per fornire delle delimitazioni superiori alla complessità
- Ω (omega grande) –per fornire delle delimitazione inferiori alla complessità
- Θ (theta) per fornire delle delimitazioni strette superiori ed inferiori alla complessità

La notazione O grande

Siano $f(n)$ e $g(n)$ due funzioni non negative; $f(n) = O(g(n))$ se esistono due costanti **positive** c ed n_0 tali che $0 \leq f(n) \leq cg(n)$ per ogni $n \geq n_0$. Graficamente:



La notazione O grande

- Se $f(n) = O(g(n))$ allora $g(n)$ è un **limite asintotico superiore** per $f(n)$: $f(n)$, a meno di un fattore costante, cresce al più come $g(n)$

La notazione O grande

- Se $f(n) = O(g(n))$ allora $g(n)$ è un **limite asintotico superiore** per $f(n)$: $f(n)$, a meno di un fattore costante, cresce al più come $g(n)$
- Esempio: il limite superiore per la ricerca in un array non ordinato è $O(n)$

La notazione O grande

- Se $f(n) = O(g(n))$ allora $g(n)$ è un **limite asintotico superiore** per $f(n)$: $f(n)$, a meno di un fattore costante, cresce al più come $g(n)$
- Esempio: il limite superiore per la ricerca in un array non ordinato è $O(n)$
- O viene usata nell'analisi del costo computazionale nel caso pessimo

La notazione O grande

- Se $f(n) = O(g(n))$ allora $g(n)$ è un **limite asintotico superiore** per $f(n)$: $f(n)$, a meno di un fattore costante, cresce al più come $g(n)$
- Esempio: il limite superiore per la ricerca in un array non ordinato è $O(n)$
- O viene usata nell'analisi del costo computazionale nel caso pessimo
- $O(g(n))$ rappresenta l'insieme di funzioni definito come:

$$\mathrm{O(g(n))} = \{f(n) \mid \quad \text{esistono delle costanti } \textbf{positive } c \text{ ed } n_0 \text{ tali che } 0 \leq f(n) \leq cg(n) \quad \}$$

La notazione O grande

- Se $f(n) = O(g(n))$ allora $g(n)$ è un **limite asintotico superiore** per $f(n)$: $f(n)$, a meno di un fattore costante, cresce al più come $g(n)$
- Esempio: il limite superiore per la ricerca in un array non ordinato è $O(n)$
- O viene usata nell'analisi del costo computazionale nel caso pessimo
- $O(g(n))$ rappresenta l'insieme di funzioni definito come:

$$\mathrm{O(g(n))} = \{f(n) \mid \quad \text{esistono delle costanti } \textbf{positive } c \text{ ed } n_0 \text{ tali che } 0 \leq f(n) \leq cg(n) \quad \}$$

- Scrivere $f(n) = O(g(n))$ è un "abuso" di notazione; avremmo dovuto scrivere $f(n) \in O(g(n))$

La notazione O grande: un esempio

Example (Provare che $f(n) = 3n^2 + 10n = O(n^2)$)

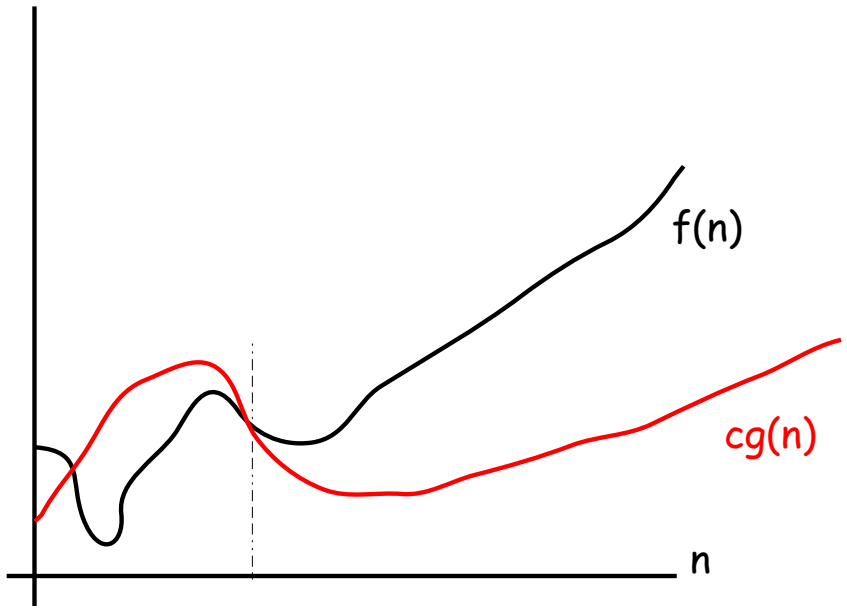
Dobbiamo dimostrare l'esistenza di costanti positive c ed n_0 tali che $0 \leq f(n) \leq cn^2$ per ogni $n \geq n_0$

Basta scegliere $c = 4$ e $n_0 = 10$; infatti:

$$
\begin{aligned}
& 3n^2 + 10n \leq cn^2 \\
& cn^2 - 3n^2 - 10n \geq 0 \\
& (c-3)n^2 - 10n \geq 0 \quad \text{scegliamo } c = 4 \\
& n^2 - 10n \geq 0 \\
& n(n-10) \geq 0 \qquad \text{vera per ogni } n \geq n_0 = 10
\end{aligned}
$$

La notazione Ω grande

Siano $f(n)$ e $g(n)$ due funzioni non negative; $f(n) = \Omega(g(n))$ se esistono due costanti **positive** c ed n_0 tali che $0 \leq cg(n) \leq f(n)$ per ogni $n \geq n_0$. Graficamente:



La notazione Ω grande

- Se $f(n) = \Omega(g(n))$ allora $g(n)$ è un **limite asintotico inferiore** per $f(n)$: $f(n)$, a meno di un fattore costante, cresce almeno come $g(n)$

La notazione Ω grande

- Se $f(n) = \Omega(g(n))$ allora $g(n)$ è un **limite asintotico inferiore** per $f(n)$: $f(n)$, a meno di un fattore costante, cresce almeno come $g(n)$
- Esempio: il limite inferiore per la ricerca in un array non ordinato è $\Omega(n)$

La notazione Ω grande

- Se $f(n) = \Omega(g(n))$ allora $g(n)$ è un **limite asintotico inferiore** per $f(n)$: $f(n)$, a meno di un fattore costante, cresce almeno come $g(n)$
- Esempio: il limite inferiore per la ricerca in un array non ordinato è $\Omega(n)$
- Ω viene usata nell'analisi del costo computazionale nel caso ottimo

La notazione Ω grande

- Se $f(n) = \Omega(g(n))$ allora $g(n)$ è un **limite asintotico inferiore** per $f(n)$: $f(n)$, a meno di un fattore costante, cresce almeno come $g(n)$
- Esempio: il limite inferiore per la ricerca in un array non ordinato è $\Omega(n)$
- Ω viene usata nell'analisi del costo computazionale nel caso ottimo
- $\Omega(g(n))$ rappresenta l'insieme di funzioni definito come:

$$\Omega(g(n)) = \{f(n) \mid \text{esistono delle costanti } \textbf{positive } c \text{ ed } n_0 \text{ tali che } 0 \leq cg(n) \leq f(n) \}$$

La notazione Ω grande

- Se $f(n) = \Omega(g(n))$ allora $g(n)$ è un **limite asintotico inferiore** per $f(n)$: $f(n)$, a meno di un fattore costante, cresce almeno come $g(n)$
- Esempio: il limite inferiore per la ricerca in un array non ordinato è $\Omega(n)$
- Ω viene usata nell'analisi del costo computazionale nel caso ottimo
- $\Omega(g(n))$ rappresenta l'insieme di funzioni definito come:

$$\Omega(g(n)) = \{f(n) \mid \text{esistono delle costanti } \textbf{positive } c \text{ ed } n_0 \text{ tali che } 0 \leq cg(n) \leq f(n) \}$$

- Scrivere $f(n) = \Omega(g(n))$ è un "abuso" di notazione; avremmo dovuto scrivere $f(n) \in \Omega(g(n))$

La notazione Ω grande: un esempio

Example (Provare che $f(n) = 3n^2 + 10n = \Omega(n^2)$)

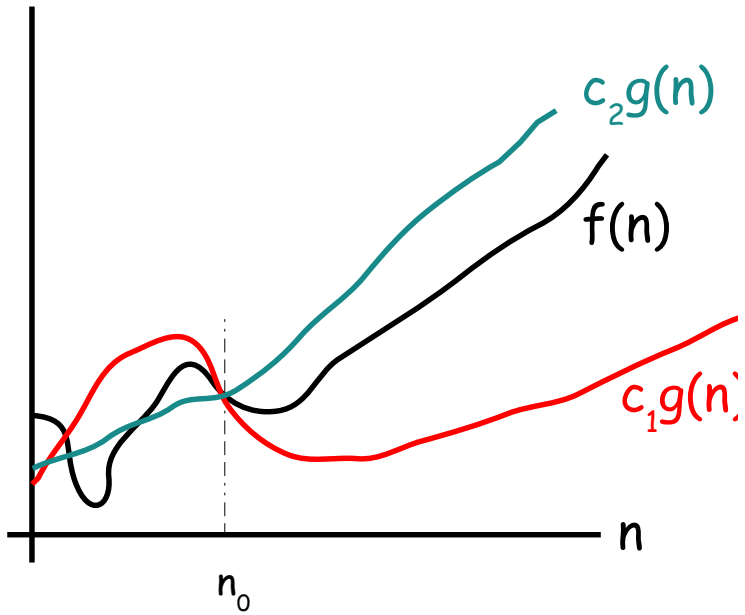
Dobbiamo dimostrare l'esistenza di costanti positive c ed n_0 tali che $0 \leq cn^2 \leq f(n)$ per ogni $n \geq n_0$

Basta scegliere $c = 3$ e $n_0 = 0$ questo perchè risulta evidente che

$$3n^2 + 10n \geq 3n^2 \text{ per ogni } n \geq n_0 = 0$$

La notazione Θ grande

Siano $f(n)$ e $g(n)$ due funzioni non negative; $f(n) = \Theta(g(n))$ se esistono tre costanti **positive** c_1, c_2 ed n_0 tali che $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ per ogni $n \geq n_0$. Graficamente:



La notazione Θ grande

- Se $f(n) = \Theta(g(n))$ allora $g(n)$ è un **limite asintotico stretto** per $f(n)$: $f(n)$, a meno di un fattore costante, cresce esattamente come $g(n)$

La notazione Θ grande

- Se $f(n) = \Theta(g(n))$ allora $g(n)$ è un **limite asintotico stretto** per $f(n)$: $f(n)$, a meno di un fattore costante, cresce esattamente come $g(n)$
- $\Theta(\mathrm{g(n)})$ rappresenta l'insieme di funzioni definito come:

$$\Theta(g(n)) = \{f(n) \mid \text{esistono delle costanti } \textbf{positive } c_1,\ c_2 \text{ ed } n_0 \text{ tali che } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \}$$

La notazione Θ grande

- Se $f(n) = \Theta(g(n))$ allora $g(n)$ è un **limite asintotico stretto** per $f(n)$: $f(n)$, a meno di un fattore costante, cresce esattamente come $g(n)$
- $\Theta(\mathrm{g(n)})$ rappresenta l'insieme di funzioni definito come:

$$\Theta(g(n)) = \{f(n) \mid \text{esistono delle costanti } \textbf{positive } c_1,\ c_2 \text{ ed } n_0 \text{ tali che } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \}$$

- Scrivere $f(n) = \Theta(g(n))$ è un "abuso" di notazione

Theorem

Per ogni coppia di funzioni $f(n)$ e $g(n)$, si ha che $f(n) = \Theta(g(n))$ se e soltanto se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$

La notazione Ω grande: un esempio

Example (Provare che $f(n) = 3n^2 + 10n = \Theta(n^2)$)

Abbiamo dimostrato che $f(n) = O(n^2)$ e $f(n) = \Omega(n^2)$. Questo basta per concludere che

$$f(n) = \Theta(n^2)$$

Le notazioni "o piccolo" ed "ω piccolo"

- Siano $f(n)$ e $g(n)$ due funzioni non negative; $f(n) = o(g(n))$ se per ogni costante **positiva** c esiste una costante $n_0 > 0$ tale che $0 \leq f(n) < cg(n)$ per ogni $n \geq n_0$

- Intuitivamente, nella notazione o, la funzione $f(n)$ diventa insignificante rispetto a $g(n)$ quando n tende all'infinito – diciamo che $g(n)$ domina $f(n)$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Le notazioni "o piccolo" ed "ω piccolo"

- Siano $f(n)$ e $g(n)$ due funzioni non negative; $f(n) = \omega(g(n))$ se per ogni costante **positiva** c esiste una costante $n_0 > 0$ tale che $0 \leq cg(n) < f(n)$ per ogni $n \geq n_0$

- Intuitivamente, nella notazione ω, la funzione $f(n)$ diventa arbitrariamente grande rispetto a $g(n)$ quando n tende all'infinito – diciamo che $g(n)$ domina $f(n)$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

Analogie con il confronto fra numeri reali

$$
\begin{aligned}
f(n) = O(g(n) &\approx a \leq b \\
f(n) = \Omega(g(n) &\approx a \geq b \\
f(n) = \Theta(g(n) &\approx a = b \\
f(n) = o(g(n) &\approx a < b \\
f(n) = \omega(g(n) &\approx a > b
\end{aligned}
$$

Transitività

$$f(n) = \Theta(g(n)) \ \wedge \ g(n) = \Theta(h(n)) \Longrightarrow f(n) = \Theta(h(n))$$

$$f(n) = O(g(n)) \ \wedge \ g(n) = O(h(n)) \Longrightarrow f(n) = O(h(n))$$

$$f(n) = \Omega(g(n)) \ \wedge \ g(n) = \Omega(h(n)) \Longrightarrow f(n) = \Omega(h(n))$$

$$f(n) = o(g(n)) \ \wedge \ g(n) = o(h(n)) \Longrightarrow f(n) = o(h(n))$$

$$f(n) = \omega(g(n)) \ \wedge \ g(n) = \omega(h(n)) \Longrightarrow f(n) = \omega(h(n))$$

Riflessività, Simmetria e Dualità

riflessività:

$f(n) = \Theta(f(n))$

$f(n) = O(f(n))$

$f(n) = \Omega(f(n))$

simmetria:

$f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n))$

dualità:

$f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$

$f(n) = o(g(n)) \iff g(n) = \omega(f(n))$